10 Service configuration with dependency injection

This chapter covers

- Understanding the benefits of dependency injection
- How ASP.NET Core uses dependency injection
- Configuring your services to work with dependency injection
- Choosing the correct lifetime for your services

In part 1 of this book, you saw the bare bones of how to build applications with ASP.NET Core. You learned how to compose middleware to create your application and how to use the MVC pattern to build traditional web applications with Razor Pages and Web APIs. This gave you the tools to start building simple applications.

In this chapter you'll see how to use *dependency injection* (DI) in your ASP.NET Core applications. DI is a design pattern that helps you develop loosely coupled code. ASP.NET Core uses the pattern extensively, both internally in the framework and in the applications you build, so you'll need to use it in all but the most trivial of applications.

You may have heard of DI before, and possibly even used it in your own applications. If so, this chapter shouldn't hold many surprises for you. If you haven't used DI before, never fear; I'll make sure you're up to speed by the time the chapter is done!

This chapter starts by introducing DI in general, the principles it drives, and why you should care about it. You'll see how ASP.NET Core has embraced DI throughout its implementation and why you should do the same when writing your own applications.

Once you have a solid understanding of the concept, you'll see how to apply DI to your own classes. You'll learn how to configure your app so that the ASP.NET Core framework can create your classes for you, removing the pain of having to create new objects manually in your code. Toward the end of the chapter, you'll learn how to control how long your objects are used for and some of the pitfalls to be aware of when you come to write your own applications.

In chapter 19, we'll look at some of the more advanced ways to use DI, including how to wire up a third-party DI container. For now, though, let's get back to basics: what is DI and why should you care about it?

10.1 Introduction to dependency injection

This section aims to give you a basic understanding of what dependency injection is, why you should care about it, and how ASP.NET Core uses it. The topic itself extends far beyond the reach of this single chapter. If you want a deeper background, I highly recommend checking out Martin Fowler's articles online.[1]

TIP For a more directly applicable read with many examples in C#, I recommend picking up *Dependency Injection Principles, Practices, and Patterns* by Steven van Deursen and Mark Seemann (Manning, 2019).

The ASP.NET Core framework has been designed from the ground up to be modular and to adhere to "good" software engineering practices. As with anything in software, what is considered best practice varies over time, but for object-oriented programming, the SOLID[2] principles have held up well.

On that basis, ASP.NET Core has *dependency injection* (sometimes called *dependency inversion*, *DI*, or *inversion of control*[3]) baked into the heart of the framework. Whether or not you want to use it within your own application code, the framework libraries themselves depend on it as a concept.

I begin this section by starting with a common scenario: a class in your application depends on a different class, which in turn depends on another. You'll see how dependency injection can help alleviate this chaining of dependencies for you and provide a number of extra benefits.

[1] Martin Fowler's website at https://martinfowler.com is a goldmine of best-practice goodness. One of the most applicable articles to this chapter can be found at www.martinfowler.com/articles/injection.html.

[2] SOLID is a mnemonic for single responsibility, open-closed, Liskov substitution, interface segregation, and dependency inversion: https://en.wikipedia.org/wiki/SOLID_(object-oriented_design).

[3] Although related, dependency injection and dependency inversion are two different things. I cover both in a general sense in this chapter, but for a good explanation of the differences, see this post by Derick Bailey titled "Dependency Injection Is NOT the Same as the Dependency Inversion Principle": http://mng.bz/5jvB.

10.1.1 Understanding the benefits of dependency injection

When you first started programming, the chances are you didn't immediately use a DI framework. That's not surprising, or even a bad thing; DI adds a certain amount of extra wiring that's often not warranted in simple applications or when you're getting started. But when things start to get more complex, DI comes into its own as a great tool to help keep that complexity in check.

Let's consider a simple example, written without any sort of DI. Imagine a user has registered on your web app and you want to send them an email. This listing shows how you might approach this initially in an API controller.

> **NOTE** I'm using an API controller for this example, but I could just as easily have used a Razor Page. Razor Pages and API controllers both use constructor dependency injection, as you'll see in section 10.2.

Listing 10.1 Sending an email without DI when there are no dependencies

```
public class UserController : ControllerBase
{
    [HttpPost("register")]
    public IActionResult RegisterUser(string username)
    {
        var emailSender = new EmailSender();
        emailSender.SendEmail(username);
        return Ok();
    }
}
```

The action method is called when a new user is created.

Creates a new instance of EmailSender

Uses the new instance to send the email

In this example, the `RegisterUser` action on `UserController` executes when a new user registers on your app. This creates a new instance of an `EmailSender` class and calls `SendEmail()` to send the email. The `EmailSender` class is the one that does the sending of the email. For the purposes of this example, you can imagine it looks something like this:

```
public class EmailSender
{
    public void SendEmail(string username)
    {
        Console.WriteLine($"Email sent to {username}!");
    }
}
```

`Console.Writeline` stands in here for the real process of sending the email.

> **NOTE** Although I'm using sending email as a simple example, in practice you might want to move this code out of your Razor Page and controller classes entirely. This type of asynchronous task is well suited to using message queues and a background process. For more details, see http://mng.bz/pVWR.

If the EmailSender class is as simple as the previous example and it has no dependencies, you might not see any need to adopt a different approach to creating objects. And to an extent, you'd be right. But what if you later update your implementation of EmailSender so that it doesn't implement the whole email-sending logic itself?

In practice, EmailSender would need to do many things to send an email. It would need to

- Create an email message
- Configure the settings of the email server
- Send the email to the email server

Doing all of that in one class would go against the single responsibility principle (SRP), so you'd likely end up with EmailSender depending on other services. Figure 10.1 shows how this web of dependencies might look. UserController wants to send an email using EmailSender, but to do so, it also needs to create the MessageFactory, NetworkClient, and EmailServerSettings objects that EmailSender depends on.

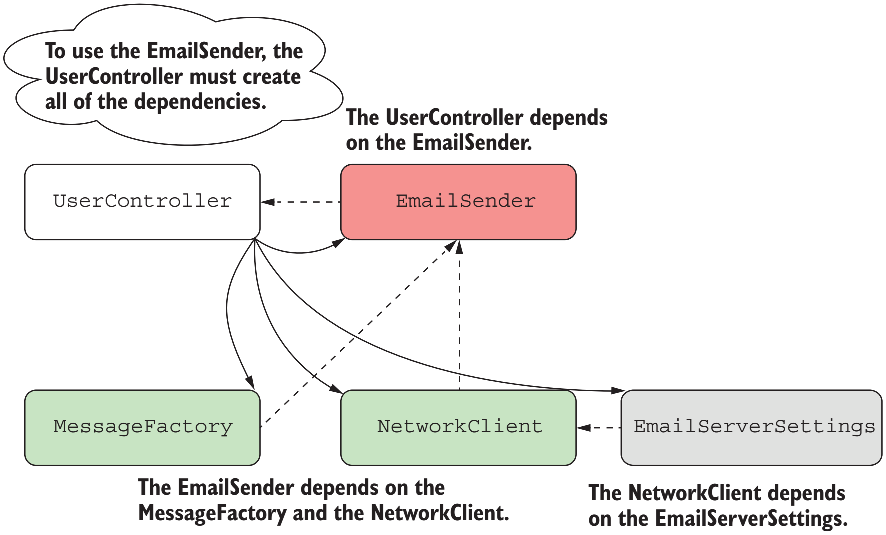

To use the EmailSender, the UserController must create all of the dependencies.
The UserController depends on the EmailSender.
UserController
EmailSender
MessageFactory
NetworkClient
EmailServerSettings
The EmailSender depends on the MessageFactory and the NetworkClient.
The NetworkClient depends on the EmailServerSettings.

Figure 10.1 Dependency diagram without dependency injection. UserController indirectly depends on all the other classes, so it has to create them all.

Each class has a number of dependencies, so the "root" class, in this case UserController, needs to know how to create every class it depends on, as well as every class its *dependencies* depend on. This is sometimes called the *dependency graph*.

DEFINITION The *dependency graph* is the set of objects that must be created in order to create a specific requested "root" object.

EmailSender depends on the MessageFactory and NetworkClient objects, so they're provided via the constructor, as shown here.

Listing 10.2 A service with multiple dependencies

```
public class EmailSender
{
    private readonly NetworkClient _client;
    private readonly MessageFactory _factory;
    public EmailSender(MessageFactory factory, NetworkClient client)
    {
        _factory = factory;
        _client = client;
    }
    public void SendEmail(string username)
    {
        var email = _factory.Create(username);
        _client.SendEmail(email);
        Console.WriteLine($"Email sent to {username}!");
    }
}
```

The EmailSender now depends on two other classes.

Instances of the dependencies are provided in the constructor.

The EmailSender coordinates the dependencies to create and send an email.

On top of that, the NetworkClient class that EmailSender depends on also has a dependency on an EmailServerSettings object:

```
public class NetworkClient
{
    private readonly EmailServerSettings _settings;
    public NetworkClient(EmailServerSettings settings)
    {
        _settings = settings;
    }
}
```

This might feel a little contrived, but it's common to find this sort of chain of dependencies. In fact, if you *don't* have this in your code, it's probably a sign that your classes are too big and aren't following the single responsibility principle.

So, how does this affect the code in UserController? The following listing shows how you now have to send an email, if you stick to new-ing up objects in the controller.

Listing 10.3 Sending email without DI when you manually create dependencies

```
public IActionResult RegisterUser(string username)
{
    var emailSender = new EmailSender(
        new MessageFactory(),
        new NetworkClient(
            new EmailServerSettings
            (
                host: "smtp.server.com",
                port: 25
            ))
```

To create EmailSender, you must create all of its dependencies.

You need a new MessageFactory.

The NetworkClient also has dependencies.

You're already two layers deep, but there could feasibly be more.

```
        );
    emailSender.SendEmail(username);
    return Ok();
}
```

Finally, you can send the email.

This is turning into some gnarly code. Improving the design of `EmailSender` to separate out the different responsibilities has made calling it from `UserController` a real chore. This code has several issues:

- *Not obeying the single responsibility principle*—Our code is now responsible for both *creating* an `EmailSender` object and *using* it to send an email.
- *Considerable ceremony*—Of the 11 lines of code in the `RegisterUser` method, only the last two are doing anything useful. This makes it harder to read and harder to understand the intent of the method.
- *Tied to the implementation*—If you decide to refactor `EmailSender` and add another dependency, you'd need to update every place it's used. Likewise, if any of the *dependencies* are refactored, you would need to update this code too.

`UserController` has an *implicit* dependency on the `EmailSender` class, as it manually creates the object itself as part of the `RegisterUser` method. The only way to know that `UserController` uses `EmailSender` is to look at its source code. In contrast, `EmailSender` has *explicit* dependencies on `NetworkClient` and `MessageFactory`, which must be provided in the constructor. Similarly, `NetworkClient` has an *explicit* dependency on the `EmailServerSettings` class.

TIP Generally speaking, any dependencies in your code should be explicit, not implicit. Implicit dependencies are hard to reason about and difficult to test, so you should avoid them wherever you can. DI is useful for guiding you along this path.

Dependency injection aims to solve the problem of building a dependency graph by *inverting* the chain of dependencies. Instead of the `UserController` creating its dependencies manually, deep inside the implementation details of the code, an already-created instance of `EmailSender` is injected via the constructor.

Now, obviously *something* needs to create the object, so the code to do that has to live somewhere. The service responsible for creating an object is called a *DI container* or an *IoC container*, as shown in figure 10.2.

DEFINITION The *DI container* or *IoC container* is responsible for creating instances of services. It knows how to construct an instance of a service by creating all its dependencies and passing these to the constructor. I'll refer to it as a DI container throughout this book.

The term *dependency injection* is often used interchangeably with *inversion of control* (IoC). DI is a specific version of the more general principle of IoC. IoC describes the

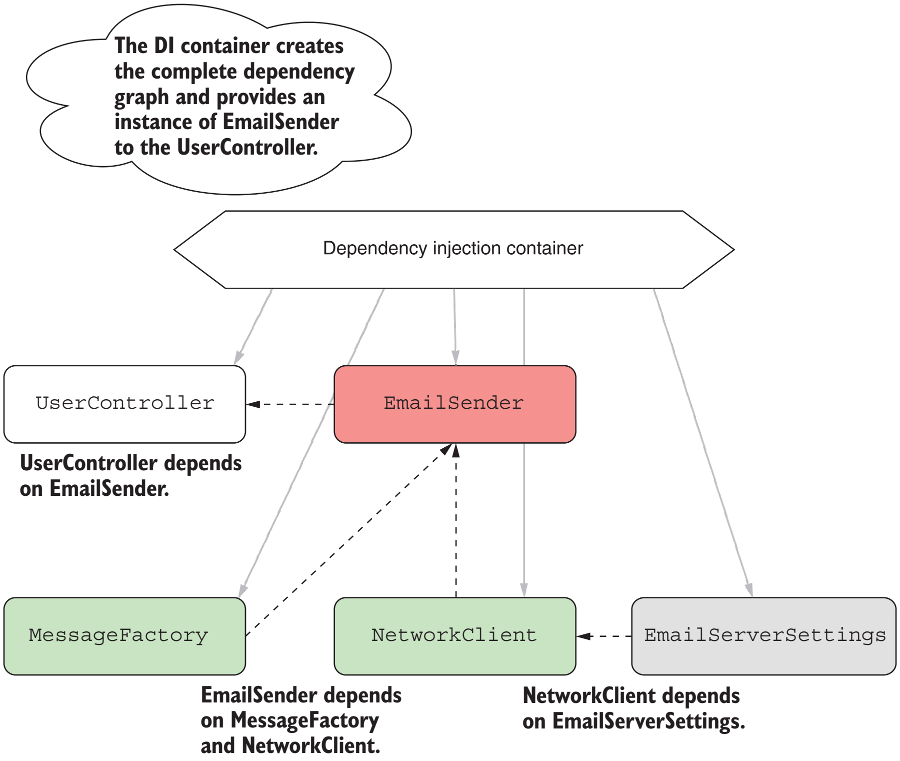

The DI container creates the complete dependency graph and provides an instance of EmailSender to the UserController.
Dependency injection container
UserController
EmailSender
UserController depends on EmailSender.
MessageFactory
NetworkClient
EmailServerSettings
EmailSender depends on MessageFactory and NetworkClient.
NetworkClient depends on EmailServerSettings.

Figure 10.2 Dependency diagram using dependency injection. `UserController` **indirectly depends on all the other classes but doesn't need to know how to create them.** `UserController` **declares that it requires** `EmailSender`**, and the container provides it.**

pattern where the *framework* calls your code to handle a request, instead of you writing the code to parse the request from bytes on the network card yourself. DI takes this further, where you allow the framework to create your dependencies too: instead of your `UserController` controlling how to create an `EmailSender` instance, it's provided one by the framework instead.

> **NOTE** Many DI containers are available for .NET: Autofac, Lamar, Unity, Ninject, Simple Injector… The list goes on! In chapter 19 you'll see how to replace the default ASP.NET Core container with one of these alternatives.

The advantage of adopting this pattern becomes apparent when you see how much it simplifies using dependencies. The following listing shows how `UserController` would look if you used DI to create `EmailSender` instead of doing it manually. All of the `new` cruft has gone, and you can focus purely on what the controller is doing—calling `EmailSender` and returning an `OkResult`.

Listing 10.4 Sending an email using DI to inject dependencies

```
public class UserController : ControllerBase
{
    private readonly EmailSender _emailSender;
    public UserController(EmailSender emailSender)
    {
        _emailSender = emailSender;
    }

    [HttpPost("register")]
    public IActionResult RegisterUser(string username)
    {
        _emailSender.SendEmail(username);
        return Ok();
    }
}
```

Instead of creating the dependencies implicitly, they're injected via the constructor.

The action method is easy to read and understand again.

One of the advantages of a DI container is that it has a single responsibility: creating objects or services. You ask a container for an instance of a service and it takes care of figuring out how to create the dependency graph, based on how you configure it.

> **NOTE** It's common to refer to *services* when talking about DI containers, which is slightly unfortunate as it's one of the most overloaded terms in software engineering! In this context, a service refers to any class or interface that the DI container creates when required.

The beauty of this approach is that by using explicit dependencies, you never have to write the mess of code you saw in listing 10.3. The DI container can inspect your service's constructor and work out how to write much of the code itself. DI containers are always configurable, so if you *want* to describe how to manually create an instance of a service you can, but by default you shouldn't need to.

> **TIP** You can inject dependencies into a service in other ways; for example, by using property injection. But constructor injection is the most common and is the only one supported out of the box in ASP.NET Core, so I'll only be using that in this book.

Hopefully the advantages of using DI in your code are apparent from this quick example, but DI provides additional benefits that you get for free. In particular, it helps keep your code loosely coupled by coding to interfaces.

10.1.2 Creating loosely coupled code

Coupling is an important concept in object-oriented programming. It refers to how a given class depends on other classes to perform its function. Loosely coupled code doesn't need to know a lot of details about a particular component to use it.

The initial example of `UserController` and `EmailSender` was an example of tight coupling; you were creating the `EmailSender` object directly and needed to know exactly how to wire it up. On top of that, the code was difficult to test. Any attempts to test UserController would result in an email being sent. If you were testing the controller with a suite of unit tests, that would be a surefire way to get your email server blacklisted for spam!

Taking EmailSender as a constructor parameter and removing the responsibility of creating the object helps reduce the coupling in the system. If the EmailSender implementation changes so that it has another dependency, you no longer have to update UserController at the same time.

One issue that remains is that UserController is still tied to an *implementation* rather than an *interface*. Coding to interfaces is a common design pattern that helps further reduce the coupling of a system, as you're not tied to a single implementation. This is particularly useful in making classes testable, as you can create "stub" or "mock" implementations of your dependencies for testing purposes, as shown in figure 10.3.

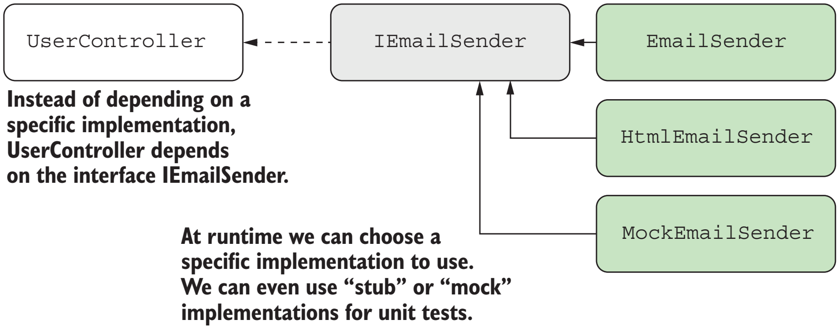

UserController
IEmailSender
EmailSender
HtmlEmailSender
MockEmailSender
Instead of depending on a specific implementation, UserController depends on the interface IEmailSender.
At runtime we can choose a specific implementation to use. We can even use "stub" or "mock" implementations for unit tests.

Figure 10.3 By coding to interfaces instead of an explicit implementation, you can use different IEmailSender implementations in different scenarios, such as a MockEmailSender in unit tests.

TIP You can choose from many different mocking frameworks. My favorite is Moq, but NSubstitute and FakeItEasy are also popular options.

As an example, you might create an IEmailSender interface, which EmailSender would implement:

```
public interface IEmailSender
{
    public void SendEmail(string username);
}
```

UserController could then depend on this interface instead of the specific EmailSender implementation, as shown in the following listing. That would allow you to use a different implementation during unit tests, such as a DummyEmailSender.

Listing 10.5 Using interfaces with dependency injection

```
public class UserController : ControllerBase
{
    private readonly IEmailSender _emailSender;
    public UserController(IEmailSender emailSender)
    {
        _emailSender = emailSender;
    }

    [HttpPost("register")]
       public IActionResult RegisterUser(string username)
    {
        _emailSender.SendEmail(username);
        return Ok();
    }
}
```

You now depend on IEmailSender instead of the specific EmailSender implementation.

You don't care what the implementation is, as long as it implements IEmailSender.

The key point here is that the consuming code, `UserController`, doesn't care how the dependency is implemented, only that it implements the `IEmailSender` interface and exposes a `SendEmail` method. The application code is now independent of the implementation.

Hopefully the principles behind DI seem sound—by having loosely coupled code, it's easy to change or swap out implementations completely. But this still leaves you with a question: how does the application know to use `EmailSender` in production instead of `DummyEmailSender`? The process of telling your DI container "when you need `IEmailSender`, use `EmailSender`" is called *registration*.

> **DEFINITION** You *register* services with a DI container so that it knows which implementation to use for each requested service. This typically takes the form of "for interface X, use implementation Y."

Exactly how you register your interfaces and types with a DI container can vary depending on the specific DI container implementation, but the principles are generally all the same. ASP.NET Core includes a simple DI container out of the box, so let's look at how it's used during a typical request.

10.1.3 Dependency injection in ASP.NET Core

ASP.NET Core was designed from the outset to be modular and composable, with an almost plugin-style architecture, which is generally complemented by DI. Consequently, ASP.NET Core includes a simple DI container that all the framework libraries use to register themselves and their dependencies.

This container is used, for example, to register the Razor Pages and Web API infrastructure—the formatters, the view engine, the validation system, and so on. It's only a basic container, so it only exposes a few methods for registering services, but you can also replace it with a third-party DI container. This can give you extra capabilities, such as auto-registration or setter injection. The DI container is built into the ASP.NET Core hosting model, as shown in figure 10.4.

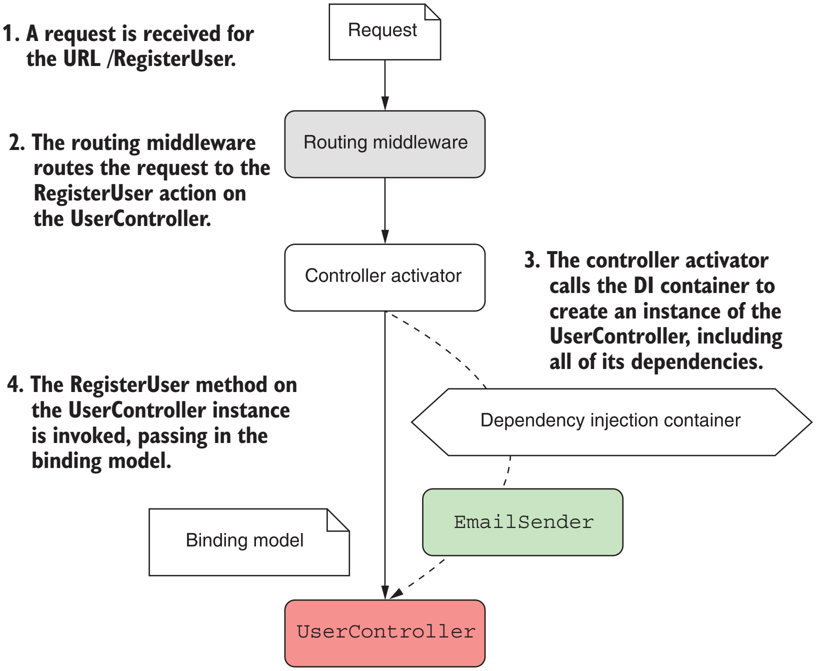

1. A request is received for the URL /RegisterUser.
Request
2. The routing middleware routes the request to the RegisterUser action on the UserController.
Routing middleware
Controller activator
3. The controller activator calls the DI container to create an instance of the UserController, including all of its dependencies.
4. The RegisterUser method on the UserController instance is invoked, passing in the binding model.
Dependency injection container
EmailSender
Binding model
UserController

Figure 10.4 The ASP.NET Core hosting model uses the DI container to fulfill dependencies when creating controllers.

The hosting model pulls dependencies from the DI container when they're needed. If the framework determines that `UserController` is required due to the incoming URL/route, the controller activator responsible for creating an API controller instance will ask the DI container for an `IEmailSender` implementation.

> **NOTE** This approach, where a class calls the DI container directly to ask for a class is called the *service locator* pattern. Generally speaking, you should try to avoid this pattern in your code; include your dependencies as constructor arguments directly and let the DI container provide them for you.[4]

The DI container needs to know what to create when asked for `IEmailSender`, so you must have registered an implementation, such as `EmailSender`, with the container. Once an implementation is registered, the DI container can inject it anywhere. That

[4] You can read about the Service Locator antipattern in *Dependency Injection Principles, Practices, and Patterns* by Steven van Deursen and Mark Seemann (Manning, 2019): http://mng.bz/6g4o.

means you can inject framework-related services into your own custom services, as long as they are registered with the container. It also means you can register alternative versions of framework services and have the framework automatically use those in place of the defaults.

The flexibility to choose exactly how and which components you combine in your applications is one of the selling points of DI. In the next section, you'll learn how to configure DI in your own ASP.NET Core application, using the default, built-in container.

10.2 Using the dependency injection container

In previous versions of ASP.NET, using dependency injection was entirely optional. In contrast, to build all but the most trivial ASP.NET Core apps, some degree of DI is required. As I've mentioned, the underlying framework depends on it, so things like using Razor Pages and API controllers require you to configure the required services.

In this section you'll see how to register these framework services with the built-in container, as well as how to register your own services. Once services are registered, you can use them as dependencies and inject them into any of the services in your application.

10.2.1 Adding ASP.NET Core framework services to the container

As I described earlier, ASP.NET Core uses DI to configure its internal components as well as your own custom services. To use these components at runtime, the DI container needs to know about all the classes it will need. You register these in the `ConfigureServices` method of your `Startup` class.

> **NOTE** The dependency injection container is set up in the `ConfigureServices` method of your `Startup` class in Startup.cs.

Now, if you're thinking, "Wait, I have to configure the internal components myself?" then don't panic. Although true in one sense—you do need to explicitly register the components with the container in your app—all the libraries you'll use expose handy extension methods to take care of the nitty-gritty details for you. These extension methods configure everything you'll need in one fell swoop, instead of leaving you to manually wire everything up.

For example, the Razor Pages framework exposes the `AddRazorPages()` extension method that you saw in chapters 2, 3, and 4. Invoke the extension method in `ConfigureServices` of `Startup`.

Listing 10.6 Registering the MVC services with the DI container

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
}
```

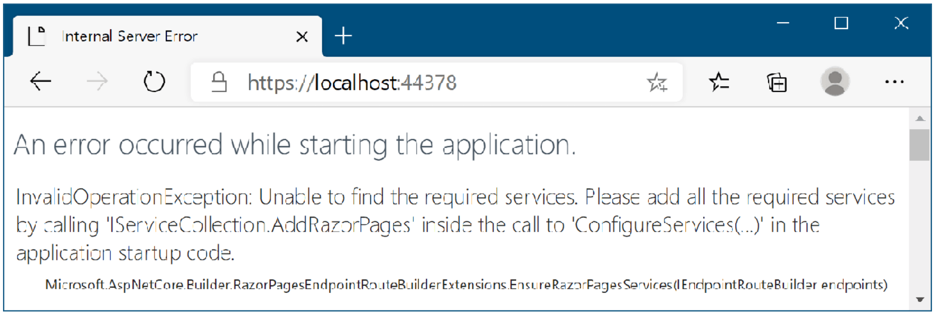
The AddRazorPages extension method adds all necessary services to the IServiceCollection.

It’s as simple as that. Under the hood, this call is registering multiple components with the DI container, using the same APIs you’ll see shortly for registering your own services.

> **TIP** The `AddControllers()` method registers the required services for API controllers, as you saw in chapter 9. There is a similar method, `AddControllersWithViews()` if you’re using MVC controllers with Razor views, and an `AddMvc()` method to add all of them and the kitchen sink!

Most nontrivial libraries that you add to your application will have services that you need to add to the DI container. By convention, each library that has necessary services should expose an `Add*()` extension method that you can call in `ConfigureServices`.

There’s no way of knowing exactly which libraries will require you to add services to the container; it’s generally a case of checking the documentation for any libraries you use. If you forget to add them, you may find the functionality doesn’t work, or you might get a handy exception like the one shown in figure 10.5. Keep an eye out for these and be sure to register any services that you need.



An error occurred while starting the application.

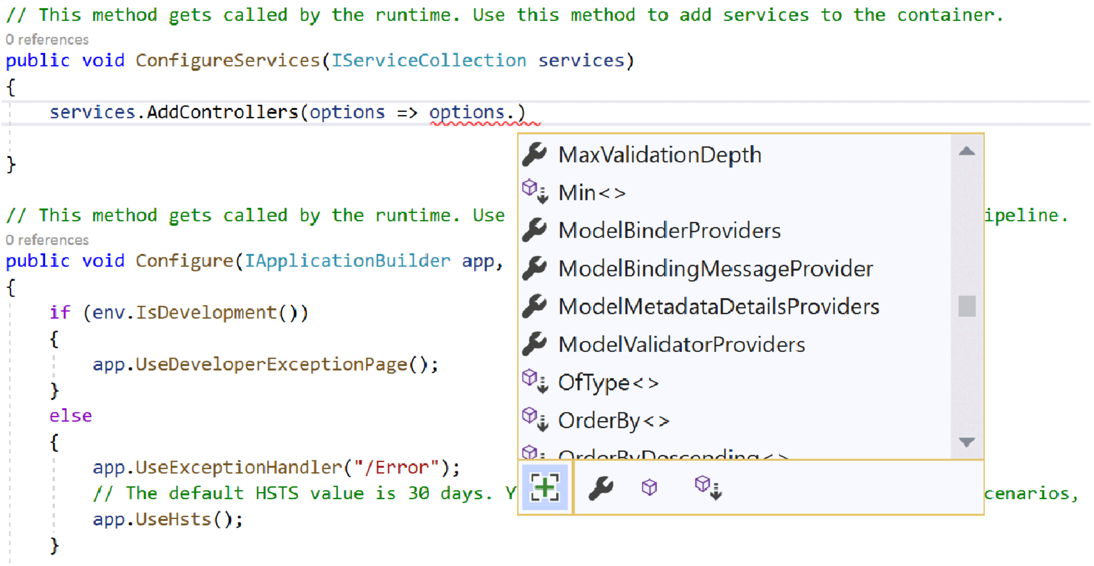
InvalidOperationException: Unable to find the required services. Please add all the required services by calling 'IServiceCollection.AddRazorPages' inside the call to 'ConfigureServices(...)' in the application startup code.

Microsoft.AspNetCore.Builder.RazorPagesEndpointRouteBuilderExtensions.EnsureRazorPagesServices(IEndpointRouteBuilder endpoints)

Figure 10.5 If you fail to call `AddRazorPages` in the `ConfigureServices` of `Startup`, you’ll get a friendly exception message at runtime.

It’s also worth noting that some of the `Add*()` extension methods allow you to specify additional options when you call them, often by way of a lambda expression. You can think of these as configuring the installation of a service into your application. The `AddControllers` method, for example, provides a wealth of options for fine-tuning its behavior if you want to get your fingers dirty, as shown by the IntelliSense snippet in figure 10.6.

Once you’ve added the required framework services, you can get down to business and register your own services, so you can use DI in your own code.



```
// This method gets called by the runtime. Use this method to add services to the container.
0 references
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(options => options.)

}

// This method gets called by the runtime. Use                                          ipeline.
0 references
public void Configure(IApplicationBuilder app,
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        // The default HSTS value is 30 days. Y                                         cenarios,
        app.UseHsts();
    }
```

Figure 10.6 Configuring services when adding them to the service collection. The `AddControllers()` function allows you to configure a wealth of the internals of the API controller services. Similar configuration options are available in the `AddRazorPages()` function.

10.2.2 Registering your own services with the container

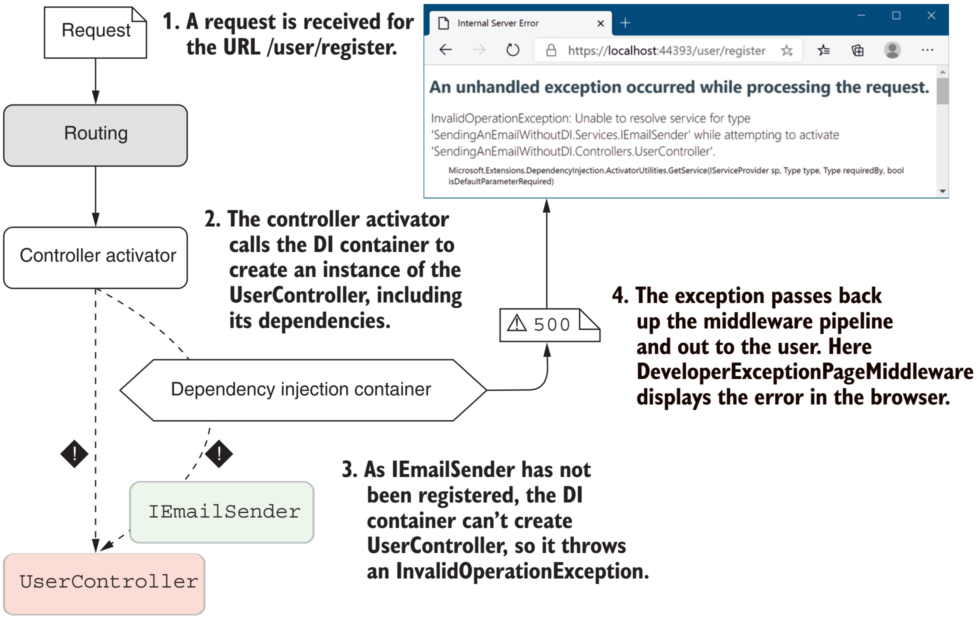
In the first section of this chapter, I described a system for sending emails when a new user registers on your application. Initially, `UserController` was manually creating an instance of `EmailSender`, but you subsequently refactored this, so you inject an instance of `IEmailSender` into the constructor instead.

The final step to make this refactoring work is to configure your services with the DI container. This lets the DI container know what to use when it needs to fulfill the `IEmailSender` dependency. If you don't register your services, you'll get an exception at runtime, like the one in figure 10.7. Luckily, this exception is useful, letting you know which service wasn't registered (`IEmailSender`) and which service needed it (`UserController`).

In order to completely configure the application, you need to register `EmailSender` and all of its dependencies with the DI container, as shown in figure 10.8.

Configuring DI consists of making a series of statements about the services in your app. For example,

- When a service requires `IEmailSender`, use an instance of `EmailSender`.
- When a service requires `NetworkClient`, use an instance of `NetworkClient`.
- When a service requires `MessageFactory`, use an instance of `MessageFactory`.



An unhandled exception occurred while processing the request.

```
InvalidOperationException: Unable to resolve service for type
'SendingAnEmailWithoutDI.Services.IEmailSender' while attempting to activate
'SendingAnEmailWithoutDI.Controllers.UserController'.
    Microsoft.Extensions.DependencyInjection.ActivatorUtilities.GetService(IServiceProvider sp, Type type, Type requiredBy, bool
    isDefaultParameterRequired)
```

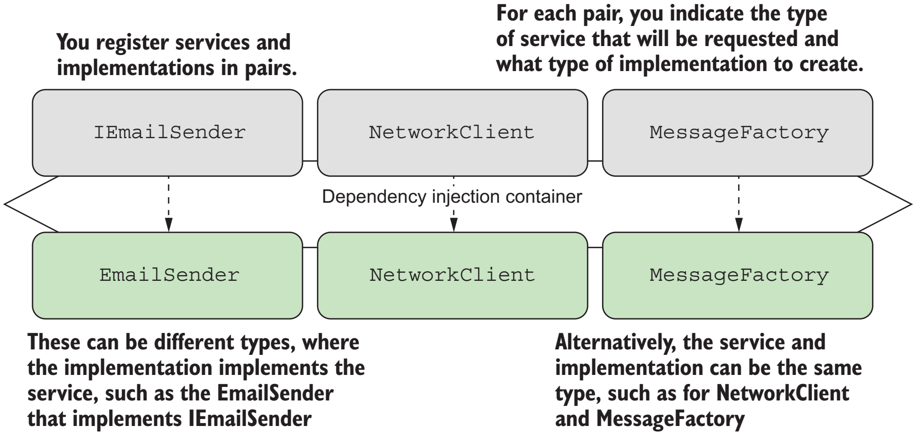
1. A request is received for the URL /user/register.

2. The controller activator calls the DI container to create an instance of the UserController, including its dependencies.

3. As IEmailSender has not been registered, the DI container can't create UserController, so it throws an InvalidOperationException.

4. The exception passes back up the middleware pipeline and out to the user. Here DeveloperExceptionPageMiddleware displays the error in the browser.

Figure 10.7 If you don't register all your required dependencies in `ConfigureServices`, you'll get an exception at runtime, telling you which service wasn't registered.



You register services and implementations in pairs.

For each pair, you indicate the type of service that will be requested and what type of implementation to create.

These can be different types, where the implementation implements the service, such as the EmailSender that implements IEmailSender

Alternatively, the service and implementation can be the same type, such as for NetworkClient and MessageFactory

Figure 10.8 Configuring the DI container in your application involves telling it what type to use when a given service is requested; for example, "Use `EmailSender` when `IEmailSender` is required."

NOTE You'll also need to register the `EmailServerSettings` object with the DI container—we'll do that slightly differently in the next section.

These statements are made by calling various `Add*` methods on `IServiceCollection` in the `ConfigureServices` method. Each method provides three pieces of information to the DI container:

- *Service type*—`TService`. This is the class or interface that will be requested as a dependency. It's often an interface, such as `IEmailSender`, but sometimes a concrete type, such as `NetworkClient` or `MessageFactory`.
- *Implementation type*—`TService` or `TImplementation`. This is the class the container should create to fulfill the dependency. It must be a concrete type, such as `EmailSender`. It may be the same as the service type, as for `NetworkClient` and `MessageFactory`.
- *Lifetime—transient, singleton, or scoped*. This defines how long an instance of the service should be used. I'll discuss lifetimes in detail in section 10.3.

The following listing shows how you can configure `EmailSender` and its dependencies in your application using three different methods: `AddScoped<TService>`, `AddSingleton<TService>`, and `AddScoped<TService, TImplementation>`. This tells the DI container how to create each of the `TService` instances when they're required.

Listing 10.7 Registering services with the DI container

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddScoped<IEmailSender, EmailSender>();
    services.AddScoped<NetworkClient>();
    services.AddSingleton<MessageFactory>();
}
```

You're using API controllers, so you must call AddControllers.

Whenever you require an IEmailSender, use EmailSender.

Whenever you require a NetworkClient, use NetworkClient.

Whenever you require a MessageFactory, use MessageFactory.

That's all there is to dependency injection! It may seem a little bit like magic,[5] but you're just giving the container instructions on how to make all the constituent parts. You give it a recipe for how to cook the chili, shred the lettuce, and grate the cheese, so that when you ask for a burrito, it can put all the parts together and hand you your meal!

The service type and implementation type are the same for `NetworkClient` and `MessageFactory`, so there's no need to specify the same type twice in the `AddScoped` method, and hence the slightly simpler signature.

[5] Under the hood, the built-in ASP.NET Core DI container uses optimized reflection to create dependencies, but different DI containers may use other approaches.

NOTE The EmailSender instance is *only* registered as an IEmailSender, so you can't retrieve it by requesting the specific EmailSender implementation; you *must* use the IEmailSender interface.

These generic methods aren't the only way to register services with the container. You can also provide objects directly or by using lambdas, as you'll see in the next section.

10.2.3 Registering services using objects and lambdas

As I mentioned earlier, I didn't *quite* register all the services required by UserController. In all my previous examples, NetworkClient depends on EmailServerSettings, which you'll also need to register with the DI container for your project to run without exceptions.

I avoided registering this object in the preceding example because you have to use a slightly different approach. The preceding Add* methods use generics to specify the Type of the class to register, but they don't give any indication of *how* to construct an instance of that type. Instead, the container makes a number of assumptions that you have to adhere to:

- The class must be a concrete type.
- The class must only have a single "valid" constructor that the container can use.
- For a constructor to be "valid," all constructor arguments must be registered with the container or they must be arguments with a default value.

NOTE These limitations apply to the simple built-in DI container. If you choose to use a third-party container in your app, it may have a different set of limitations.

The EmailServerSettings class doesn't meet these requirements, as it requires you to provide a host and port in the constructor, which are strings without default values:

```
public class EmailServerSettings
{
    public EmailServerSettings(string host, int port)
    {
        Host = host;
        Port = port;
    }
    public string Host { get; }
    public int Port { get; }
}
```

You can't register these primitive types in the container; it would be weird to say, "For every string constructor argument, in any type, use the "smtp.server.com" value."

Instead, you can create an instance of the EmailServerSettings object yourself and provide *that* to the container, as shown next. The container uses the preconstructed object whenever an instance of the EmailServerSettings object is required.

Listing 10.8 Providing an object instance when registering services

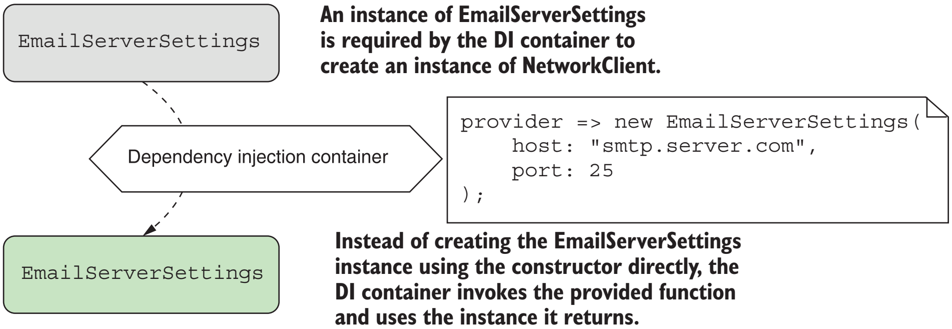
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddScoped<IEmailSender, EmailSender>();
    services.AddSingleton<NetworkClient>();
    services.AddScoped<MessageFactory>();
    services.AddSingleton(
        new EmailServerSettings
        (
            host: "smtp.server.com",
            port: 25
        ));
}
```

This instance of EmailServerSettings will be used whenever an instance is required.

This works fine if you only want to have a single instance of `EmailServerSettings` in your application—*the same object* will be shared everywhere. But what if you want to create a *new* object each time one is requested?

> **NOTE** When the same object is used whenever it's requested, it's known as a *singleton*. If you create an object and pass it to the container, it's *always* registered as a singleton. You can also register any class using the `AddSingleton<T>()` method, and the container will only use one instance throughout your application. I discuss singletons along with other *lifetimes* in detail in section 10.3. The lifetime is how long the DI container should use a given object to fulfill a service's dependencies.

Instead of providing a single instance that the container will always use, you can also provide a *function* that the container invokes when it needs an instance of the type, as shown in figure 10.9.



An instance of EmailServerSettings is required by the DI container to create an instance of NetworkClient.

```
provider => new EmailServerSettings(
    host: "smtp.server.com",
    port: 25
);
```

Instead of creating the EmailServerSettings instance using the constructor directly, the DI container invokes the provided function and uses the instance it returns.

Figure 10.9 You can register a function with the DI container that will be invoked whenever a new instance of a service is required.

The easiest way to do this is to use a lambda function (an anonymous delegate), in which the container creates a new `EmailServerSettings` object when it's needed.

Listing 10.9 Using a lambda factory function to register a dependency

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddScoped<IEmailSender, EmailSender>();
    services.AddSingleton<NetworkClient>();
    services.AddScoped<MessageFactory>();
    services.AddScoped(
        provider =>
            new EmailServerSettings
            (
                host: "smtp.server.com",
                port: 25
            ));
}
```

Because you're providing a function to create the object, you aren't restricted to a singleton.

The lambda is provided an instance of IService-Provider.

The constructor is called every time an EmailServerSettings object is required, instead of only once.

In this example, I've changed the lifetime of the created `EmailServerSettings` object to be *scoped* instead of *singleton* and provided a factory lambda function that returns a new `EmailServerSettings` object. Every time the container requires a new `EmailServerSettings`, it executes the function and uses the new object it returns.

When you use a lambda to register your services, you're provided with an `IServiceProvider` instance at runtime, called `provider` in listing 10.9. This is the public API of the DI container itself which exposes the `GetService()` function. If you need to obtain dependencies to create an instance of your service, you can reach into the container at runtime in this way, but you should avoid doing so if possible.

TIP Avoid calling `GetService()` in your factory functions if possible. Instead, favor constructor injection—it's more performant as well as being simpler to reason about.

Open generics and dependency injection

As already mentioned, you couldn't use the generic registration methods with `EmailServerSettings` because it uses primitive dependencies (in this case, `string`) in its constructor. You also can't use the generic registration methods to register *open generics*.

Open generics are types that contain a generic type parameter, such as `Repository<T>`. You normally use this sort of type to define a base behavior that you can use with multiple generic types. In the `Repository<T>` example, you might inject `IRepository<Customer>` into your services, which should inject an instance of `DbRepository<Customer>`, for example.

(continued)

To register these types, you must use a different overload of the Add* methods. For example,

```
services.AddScoped(typeof(IRespository<>), typeof(DbRepository<>));
```

This ensures that whenever a service constructor requires IRespository<T>, the container injects an instance of DbRepository<T>.

At this point, all your dependencies are registered. But ConfigureServices is starting to look a little messy, isn't it? It's entirely down to personal preference, but I like to group my services into logical collections and create extension methods for them, as in the following listing. This creates an equivalent of the framework's AddControllers() extension method—a nice, simple, registration API. As you add more and more features to your app, I think you'll appreciate it too.

Listing 10.10 Creating an extension method to tidy up adding multiple services

```
public static class EmailSenderServiceCollectionExtensions
{
    public static IServiceCollection AddEmailSender(
        this IServiceCollection services)
    {
        services.AddScoped<IEmailSender, EmailSender>();
        services.AddSingleton<NetworkClient>();
        services.AddScoped<MessageFactory>();
        services.AddSingleton(
            new EmailServerSettings
            (
                host: "smtp.server.com",
                port: 25
            ));
        return services;
    }
}
```

Create an extension method on IServiceCollection by using the "this" keyword.

Cut and paste your registration code from ConfigureServices.

By convention, return the IServiceCollection to allow method chaining.

With the preceding extension method created, the ConfigureServices method is much easier to grok!

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.AddEmailSender();
}
```

So far, you've seen how to register the simple DI cases where you have a single implementation of a service. In some scenarios, you might find you have multiple implementations of an interface. In the next section you'll see how to register these with the container to match your requirements.

10.2.4 Registering a service in the container multiple times

One of the advantages of coding to interfaces is that you can create multiple implementations of a service. For example, imagine you want to create a more generalized version of IEmailSender so that you can send messages via SMS or Facebook, as well as by email. You create an interface for it,

```
public interface IMessageSender
{
    public void SendMessage(string message);
}
```

as well as several implementations: EmailSender, SmsSender, and FacebookSender. But how do you register these implementations in the container? And how can you inject these implementations into your UserController? The answer varies slightly depending on whether you want to use all the implementations in your consumer or only one of them.

Injecting multiple implementations of an interface

Imagine you want to send a message using each of the IMessageSender implementations whenever a new user registers, so that they get an email, an SMS, and a Facebook message, as shown in figure 10.10.

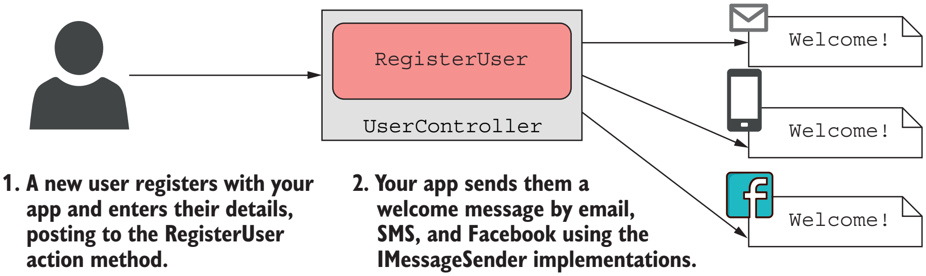
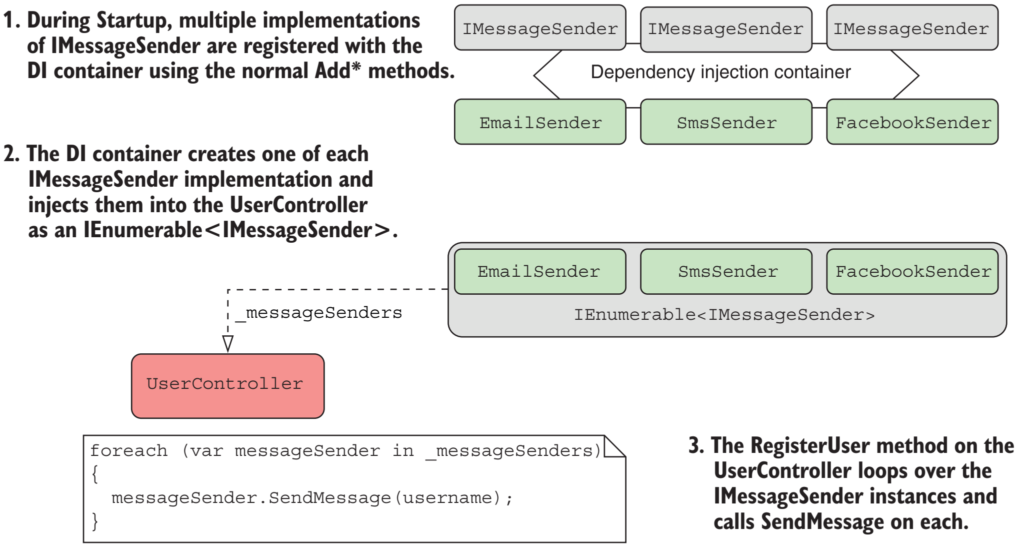

RegisterUser
UserController
Welcome!
Welcome!
Welcome!
1. A new user registers with your app and enters their details, posting to the RegisterUser action method.
2. Your app sends them a welcome message by email, SMS, and Facebook using the IMessageSender implementations.

Figure 10.10 When a user registers with your application, they call the RegisterUser method. This sends them an email, an SMS, and a Facebook message using the IMessageSender classes.

The easiest way to achieve this is to register all the service implementations in your DI container and have it inject one of each type into UserController. UserController can then use a simple foreach loop to call SendMessage() on each implementation, as in figure 10.11.



1. During Startup, multiple implementations of IMessageSender are registered with the DI container using the normal Add* methods.

2. The DI container creates one of each IMessageSender implementation and injects them into the UserController as an IEnumerable<IMessageSender>.

3. The RegisterUser method on the UserController loops over the IMessageSender instances and calls SendMessage on each.

```
foreach (var messageSender in _messageSenders)
{
  messageSender.SendMessage(username);
}
```

Figure 10.11 You can register multiple implementations of a service with the DI container, such as `IEmailSender` in this example. You can retrieve an instance of each of these implementations by requiring `IEnumerable<IMessageSender>` in the `UserController` constructor.

You register multiple implementations of the same service with a DI container in exactly the same way as for single implementations, using the `Add*` extension methods. For example,

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.AddScoped<IMessageSender, EmailSender>();
    services.AddScoped<IMessageSender, SmsSender>();
    services.AddScoped<IMessageSender, FacebookSender>();
}
```

You can then inject `IEnumerable<IMessageSender>` into `UserController`, as shown in the following listing. The container injects an array of `IMessageSender` containing one of each of the implementations you have registered, in the same order as you registered them. You can then use a standard `foreach` loop in the `RegisterUser` method to call `SendMessage` on each implementation.

Listing 10.11 Injecting multiple implementations of a service into a consumer

```
public class UserController : ControllerBase
{
```

Requesting an IEnumerable will inject an array of IMessageSender.

```
    private readonly IEnumerable<IMessageSender> _messageSenders;
    public UserController(
        IEnumerable<IMessageSender> messageSenders)
    {
        _messageSenders = messageSenders;
    }

    [HttpPost("register")]
    public IActionResult RegisterUser(string username)
    {
        foreach (var messageSender in _messageSenders)
        {
            messageSender.SendMessage(username);
        }

        return Ok();
    }
}
```

Each IMessageSender in the IEnumerable is a different implementation.

> **WARNING** You must use `IEnumerable<T>` as the constructor argument to inject all the registered types of a service, `T`. Even though this will be injected as a `T[]` array, you can't use `T[]` or `ICollection<T>` as your constructor argument. Doing so will cause an `InvalidOperationException`, similar to that in figure 10.7.

It's simple enough to inject all the registered implementations of a service, but what if you only need one? How does the container know which one to use?

Injecting a single implementation when multiple services are registered

Imagine you've already registered all the `IMessageSender` implementations; what happens if you have a service that requires only one of them? For example,

```
public class SingleMessageSender
{
    private readonly IMessageSender _messageSender;
    public SingleMessageSender(IMessageSender messageSender)
    {
        _messageSender = messageSender;
    }
}
```

The container needs to pick a *single* `IMessageSender` to inject into this service, out of the three implementations available. It does this by using the *last* registered implementation—the `FacebookSender` from the previous example.

> **NOTE** The DI container will use the last registered implementation of a service when resolving a single instance of the service.

This can be particularly useful for replacing built-in DI registrations with your own services. If you have a custom implementation of a service that you know is registered within a library's `Add*` extension method, you can override that registration by registering your own implementation afterwards. The DI container will use your implementation whenever a single instance of the service is requested.

The main disadvantage with this approach is that you still end up with *multiple* implementations registered—you can inject an IEnumerable<T> as before. Sometimes you want to conditionally register a service, so you only ever have a single registered implementation.

CONDITIONALLY REGISTERING SERVICES USING TRYADD

Sometimes you'll only want to add an implementation of a service if one hasn't already been added. This is particularly useful for library authors; they can create a default implementation of an interface and only register it if the user hasn't already registered their own implementation.

You can find several extension methods for conditional registration in the Microsoft.Extensions.DependencyInjection.Extensions namespace, such as TryAddScoped. This checks to make sure a service hasn't been registered with the container before calling AddScoped on the implementation. The following listing shows how you can conditionally add SmsSender, only if there are no existing IMessageSender implementations. As you previously registered EmailSender, the container will ignore the SmsSender registration, so it won't be available in your app.

Listing 10.12 Conditionally adding a service using TryAddScoped

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IMessageSender, EmailSender>();
    services.TryAddScoped<IMessageSender, SmsSender>();
}
```

EmailSender is registered with the container.

There's already an IMessageSender implementation, so SmsSender isn't registered.

Code like this often doesn't make a lot of sense at the application level, but it can be useful if you're building libraries for use in multiple apps. The ASP.NET Core framework, for example, uses TryAdd* in many places, which lets you easily register alternative implementations of internal components in your own application if you want.

You can also *replace* a previously registered implementation by using the Replace() extension method. Unfortunately, the API for this method isn't as friendly as the TryAdd methods. To replace a previously registered IMessageSender with SmsSender, you would use

```
services.Replace(new ServiceDescriptor(
    typeof(IMessageSender), typeof(SmsSender), ServiceLifetime.Scoped
));
```

TIP When using Replace, you must provide the same lifetime as was used to register the service that is being replaced.

That pretty much covers registering dependencies. Before we look in more depth at the "lifetime" aspect of dependencies, we'll take a quick detour and look at two ways other than a constructor to inject dependencies in your app.

10.2.5 Injecting services into action methods, page handlers, and views

I mentioned in section 10.1 that the ASP.NET Core DI container only supports constructor injection, but there are three additional locations where you can use dependency injection:

- Action methods
- Page handler methods
- View templates

In this section, I'll briefly discuss these three situations, how they work, and when you might want to use them.

INJECTING SERVICES DIRECTLY INTO ACTION METHODS AND PAGE HANDLERS USING [FROMSERVICES]

API controllers typically contain multiple action methods that logically belong together. You might group all the action methods related to managing user accounts into the same controller, for example. This allows you to apply filters and authorization to all the action methods collectively, as you'll see in chapter 13.

As you add additional action methods to a controller, you may find that the controller needs additional services to implement new action methods. With constructor injection, all these dependencies are provided via the constructor. That means the DI container must create *all* the dependencies for *every* action method in a controller, even if none of them are required by the action method being called.

Consider listing 10.13 for example. This shows `UserController` with two stub methods: `RegisterUser` and `PromoteUser`. Each action method requires a different dependency, so both dependencies will be created and injected, whichever action method is called by the request. If `IPromotionService` or `IMessageSender` have lots of dependencies themselves, the DI container may have to create lots of objects for a service that is often not used.

Listing 10.13 Injecting services into a controller via the constructor

```
public class UserController : ControllerBase
{
    private readonly IMessageSender _messageSender;
    private readonly IPromotionService _promoService;
    public UserController(
        IMessageSender messageSender, IPromotionService promoService)
    {
        _messageSender = messageSender;
        _promoService = promoService;
    }

    [HttpPost("register")]
    public IActionResult RegisterUser(string username)
```

Both IMessageSender and IPromotionService are injected into the constructor every time.

```
    {
        _messageSender.SendMessage(username);
        return Ok();
    }

    [HttpPost("promote")]
    public IActionResult PromoteUser(string username, int level)
    {
        _promoService.PromoteUser(username, level);
        return Ok();
    }
}
```

The RegisterUser method only uses IMessageSender.

The PromoteUser method only uses IPromotionService.

If you know a service is particularly expensive to create, you can choose to inject it as a dependency *directly* into the action method, instead of into the controller's constructor. This ensures the DI container only creates the dependency when the *specific* action method is invoked, as opposed to when *any* action method on the controller is invoked.

> **NOTE** Generally speaking, your controllers should be sufficiently cohesive that this approach isn't necessary. If you find you have a controller that's dependent on many services, each of which is used by a single action method, you might want to consider splitting up your controller.

You can directly inject a dependency into an action method by passing it as a parameter to the method and using the [FromServices] attribute. During model binding, the framework will resolve the parameter from the DI container, instead of from the request values. This listing shows how you could rewrite listing 10.13 to use [FromServices] instead of constructor injection.

Listing 10.14 Injecting services into a controller using the [FromServices] attribute

```
public class UserController : ControllerBase
{
    [HttpPost("register")]
    public IActionResult RegisterUser(
        [FromServices] IMessageSender messageSender,
        string username)
    {
        messageSender.SendMessage(username);
        return Ok();
    }

    [HttpPost("promote")]
    public IActionResult PromoteUser(
        [FromServices] IPromotionService promoService,
        string username, int level)
    {
        promoService.PromoteUser(username, level);
        return Ok();
    }
}
```

The [FromServices] attribute ensures IMessageSender is resolved from the DI container.

IMessageSender is only available in RegisterUser.

IPromotionService is resolved from the DI container and injected as a parameter.

Only the PromoteUser method can use IPromotionService.

You might be tempted to use the [FromServices] attribute in all your action methods, but I'd encourage you to use standard constructor injection most of the time. Having the constructor as a single location that declares all the dependencies of a class can be useful, so I only use [FromServices] in the rare cases where creating an instance of a dependency is expensive and is only used in a single action method.

The [FromServices] attribute can be used in exactly the same way with Razor Pages. You can inject services into a Razor Page's page handler, instead of into the constructor, as shown in listing 10.15.

TIP Just because you *can* inject services into page handlers like this doesn't mean you *should*. Razor Pages are inherently designed to be small and cohesive, so it's better to just use constructor injection.

Listing 10.15 Injecting services into a Razor Page using the [FromServices] attribute

```
public class PromoteUserModel: PageModel
{
    public void OnGet()
    {
    }

    public IActionResult OnPost(
        [FromServices] IPromotionService promoService,
        string username, int level)
    {
        promoService.PromoteUser(username, level);
        return RedirectToPage("success");
    }

}
```

The OnGet handler does not require any services.

IPromotionService is resolved from the DI container and injected as a parameter.

Only the OnPost page handler can use IPromotionService.

Generally speaking, if you find you need to use the [FromServices] attribute, you should step back and look at your controller/Razor Page. It's likely that you're trying to do too much in one class. Instead of working around the issue with [FromServices], consider splitting the class up or pushing some behavior down into your application model services.

Injecting services into view templates

Injecting dependencies into the constructor is recommended, but what if you don't *have* a constructor? In particular, how do you go about injecting services into a Razor view template when you don't have control over how the template is constructed?

Imagine you have a simple service, HtmlGenerator, to help you generate HTML in your view templates. The question is, how do you pass this service to your view templates, assuming you've already registered it with the DI container?

One option is to inject the HtmlGenerator into your Razor Page using constructor injection and expose the service as a property on your PageModel, as you saw in chapter 7. This will often be the easiest approach, but in some cases you might not want to have references to the HtmlGenerator service in your PageModel at all. In those cases you can directly inject HtmlGenerator into your view templates.

> **NOTE** Some people take offense to injecting services into views in this way. You definitely shouldn't be injecting services related to business logic into your views, but I think it makes sense for services that are related to HTML generation.

You can inject a service into a Razor template with the @inject directive by providing the type to inject and a name for the injected service in the template.

Listing 10.16 Injecting a service into a Razor view template with @inject

```
@inject HtmlGenerator htmlHelper
<h1>The page title</h1>
<footer>
    @htmlHelper.Copyright()
</footer>
```

Injects an instance of HtmlGenerator into the view, named htmlHelper

Uses the injected service by calling the htmlHelper instance

Injecting services directly into views can be a useful way of exposing UI-related services to your view templates without having to take a dependency on the service in your PageModel. You probably won't find you need to rely on it too much, but it's a useful tool to have.

That pretty much covers registering and using dependencies, but there's one important aspect I've only vaguely touched on: lifetimes, or when does the container create a new instance of a service? Understanding lifetimes is crucial to working with DI containers, so it's important to pay close attention to them when registering your services with the container.

10.3 Understanding lifetimes: When are services created?

Whenever the DI container is asked for a particular registered service, such as an instance of IMessageSender, it can do one of two things:

- Create and return a new instance of the service
- Return an existing instance of the service

The *lifetime* of a service controls the behavior of the DI container with respect to these two options. You define the lifetime of a service during DI service registration. This dictates when a DI container will reuse an existing instance of the service to fulfill service dependencies, and when it will create a new one.

> **DEFINITION** The *lifetime* of a service is how long an instance of a service should *live* in a container before it creates a new instance.

It's important to get your head around the implications for the different lifetimes used in ASP.NET Core, so this section looks at each available lifetime option and when you should use it. In particular, you'll see how the lifetime affects how often the DI container creates new objects. In section 10.3.4, I'll show you a pattern of lifetimes to look out for, where a short-lifetime dependency is "captured" by a long-lifetime dependency. This can cause some hard-to-debug issues, so it's important to bear in mind when configuring your app.

In ASP.NET Core, you can specify three different lifetimes when registering a service with the built-in container:

- *Transient*—Every time a service is requested, a new instance is created. This means you can potentially have different instances of the same class within the same dependency graph.
- *Scoped*—Within a *scope*, all requests for a service will give you the same object. For different scopes you'll get different objects. In ASP.NET Core, each web request gets its own scope.
- *Singleton*—You'll always get the same instance of the service, no matter which scope.

> **NOTE** These concepts align well with most other DI containers, but the terminology often differs. If you're familiar with a third-party DI container, be sure you understand how the lifetime concepts align with the built-in ASP.NET Core DI container.

To illustrate the behavior of each lifetime, I'll use a simple representative example in this section. Imagine you have `DataContext`, which has a connection to a database, as shown in listing 10.17. It has a single property, `RowCount`, which displays the number of rows in the `Users` table of a database. For the purposes of this example, we emulate calling the database by setting the number of rows in the constructor, so you will get the same value every time you call `RowCount` on a given `DataContext` instance. *Different* instances of `DataContext` will return a *different* `RowCount` value.

Listing 10.17 `DataContext` generating a random `RowCount` in its constructor

```
public class DataContext
{
    static readonly Random _rand = new Random();
    public DataContext()
    {
        RowCount = _rand.Next(1, 1_000_000_000);
    }

    public int RowCount { get; }
}
```

Generates a random number between 1 and 1,000,000,000

Read-only property set in the constructor, so it always returns the same value

You also have a `Repository` class that has a dependency on the `DataContext`, as shown in the next listing. This also exposes a `RowCount` property, but this property delegates the call to its instance of `DataContext`. Whatever value `DataContext` was created with, the `Repository` will display the same value.

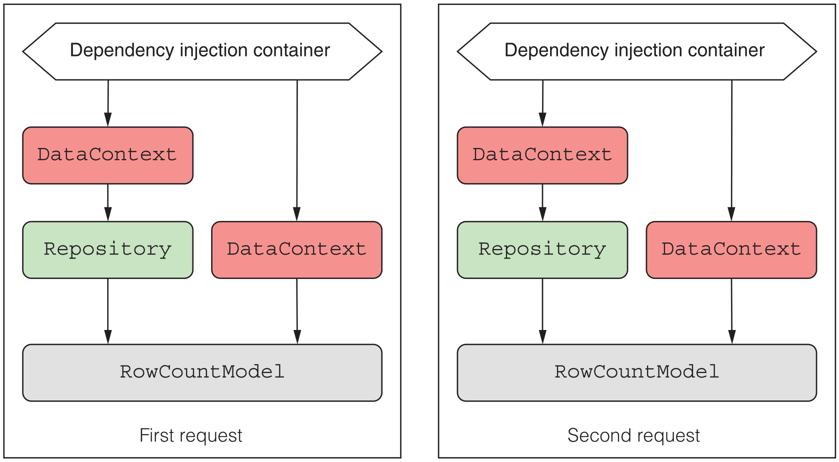
Listing 10.18 Repository service that depends on an instance of `DataContext`

```
public class Repository
{
    private readonly DataContext _dataContext;
    public Repository(DataContext dataContext)
    {
        _dataContext = dataContext;
    }
    public int RowCount => _dataContext.RowCount;
}
```

An instance of DataContext is provided using DI.

RowCount returns the same value as the current instance of DataContext.

Finally, you have the Razor Page `RowCountModel`, which takes a dependency on both `Repository` and on `DataContext` directly. When the Razor Page activator creates an instance of `RowCountModel`, the DI container injects an instance of `DataContext` and an instance of `Repository`. To create `Repository`, it also creates a second instance of `DataContext`. Over the course of two requests, a total of *four* instances of `DataContext` will be required, as shown in figure 10.12.



For each request, two instances of DataContext are required to build the RowCountModel instance.

A total of four DataContext instances are required for two requests.

Figure 10.12 The DI container uses two instances of `DataContext` for each request. Depending on the lifetime with which the `DataContext` type is registered, the container might create one, two, or four different instances of `DataContext`.

`RowCountModel` records the value of `RowCount` returned from both `Repository` and `DataContext` as properties on the `PageModel`. These are then rendered using a Razor template (not shown).

Listing 10.19 `RowCountModel` depends on `DataContext` and `Repository`

```
public class RowCountModel : PageModel
{
    private readonly Repository _repository;
    private readonly DataContext _dataContext;
    public RowCountPageModel(
        Repository repository,
        DataContext dataContext)
    {
        _repository = repository;
        _dataContext = dataContext;
    }

     public void OnGet()
    {
        DataContextCount = _dataContext.RowCount;
        RepositoryCount = _repository.RowCount;
    }

    public int DataContextCount { get; set ;}
    public int RepositoryCount { get; set ;}
}
```

DataContext and Repository are passed in using DI.

When invoked, the page handler retrieves and records RowCount from both dependencies.

The counts are exposed on the PageModel and are rendered to HTML in the Razor view.

The purpose of this example is to explore the relationship between the four `DataContext` instances, depending on the lifetimes you use to register the services with the container. I'm generating a random number in `DataContext` as a way of uniquely identifying a `DataContext` instance, but you can think of this as being a point-in-time snapshot of the number of users logged in to your site, for example, or the amount of stock in a warehouse.

I'll start with the shortest-lived lifetime, *transient*, move on to the common *scoped* lifetime, and then take a look at *singletons*. Finally, I'll show an important trap you should be on the lookout for when registering services in your own apps.

10.3.1 Transient: Everyone is unique

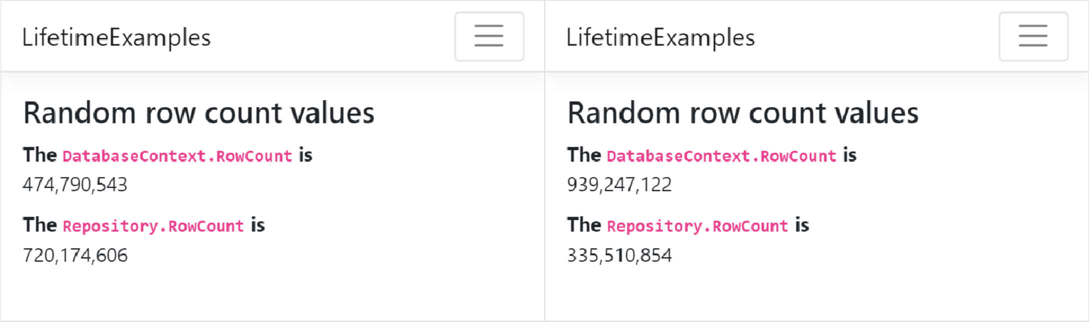
In the ASP.NET Core DI container, transient services are always created new, whenever they're needed to fulfill a dependency. You can register your services using the `AddTransient` extension methods:

```
services.AddTransient<DataContext>();
services.AddTransient<Repository>();
```

When registered in this way, every time a dependency is required the container will create a new one. This applies both *between* requests but also *within* requests; the `DataContext` injected into the `Repository` will be a different instance from the one injected into the `RowCountModel`.

> **NOTE** Transient dependencies can result in different instances of the same type within a single dependency graph.

Figure 10.13 shows the results you get from two consecutive requests when you use the transient lifetime for both services. Note that, by default, Razor Page and API controller instances are also transient and are always created anew.



LifetimeExamples

Random row count values

The `DatabaseContext.RowCount` is
474,790,543

The `Repository.RowCount` is
720,174,606

LifetimeExamples

Random row count values

The `DatabaseContext.RowCount` is
939,247,122

The `Repository.RowCount` is
335,510,854

Figure 10.13 When registered using the transient lifetime, all four `DataContext` objects are different. That can be seen by the four different numbers displayed over the course of two requests.

Transient lifetimes can result in a lot of objects being created, so they make the most sense for lightweight services with little or no state. It's equivalent to calling `new` every time you need a new object, so bear that in mind when using it. You probably won't use the transient lifetime too often; the majority of your services will probably be *scoped* instead.

10.3.2 Scoped: Let's stick together

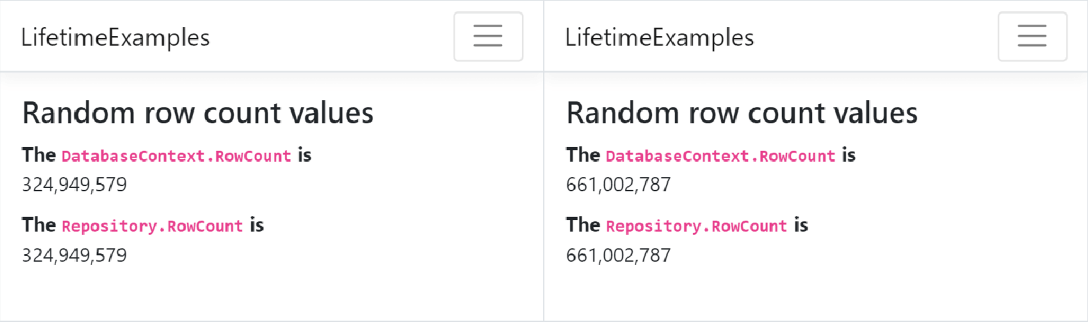
The scoped lifetime states that a *single* instance of an object will be used *within* a given scope, but a *different* instance will be used *between different scopes*. In ASP.NET Core, a scope maps to a request, so within a single request the container will use the same object to fulfill all dependencies.

For the row count example, this means that, within a single request (a single scope), the same `DataContext` will be used throughout the dependency graph. The `DataContext` injected into the `Repository` will be the same instance as that injected into `RowCountModel`.

In the next request, you'll be in a different scope, so the container will create a new instance of `DataContext`, as shown in figure 10.14. A different instance means a different `RowCount` for each request, as you can see.

You can register dependencies as scoped using the `AddScoped` extension methods. In this example, I registered `DataContext` as scoped and left `Repository` as transient, but you'd get the same results in this case if they were both scoped:

```
services.AddScoped<DataContext>();
```



LifetimeExamples

Random row count values

The DatabaseContext.RowCount is

324,949,579

The Repository.RowCount is

324,949,579

LifetimeExamples

Random row count values

The DatabaseContext.RowCount is

661,002,787

The Repository.RowCount is

661,002,787

Figure 10.14 Scoped dependencies use the same instance of `DataContext` within a single request but a new instance for a separate request. Consequently, the `RowCounts` are identical within a request.

Due to the nature of web requests, you'll often find services registered as scoped dependencies in ASP.NET Core. Database contexts and authentication services are common examples of services that should be scoped to a request—anything that you want to share across your services *within* a single request but that needs to change *between* requests.

Generally speaking, you'll find a lot of services registered using the scoped lifetime—especially anything that uses a database or is dependent on a specific request. But some services don't need to change between requests, such as a service that calculates the area of a circle or that returns the current time in different time zones. For these, a singleton lifetime might be more appropriate.

10.3.3 Singleton: There can be only one

The singleton is a pattern that came before dependency injection; the DI container provides a robust and easy-to-use implementation of it. The singleton is conceptually simple: an instance of the service is created when it's first needed (or during registration, as in section 10.2.3) and that's it. You'll always get the same instance injected into your services.

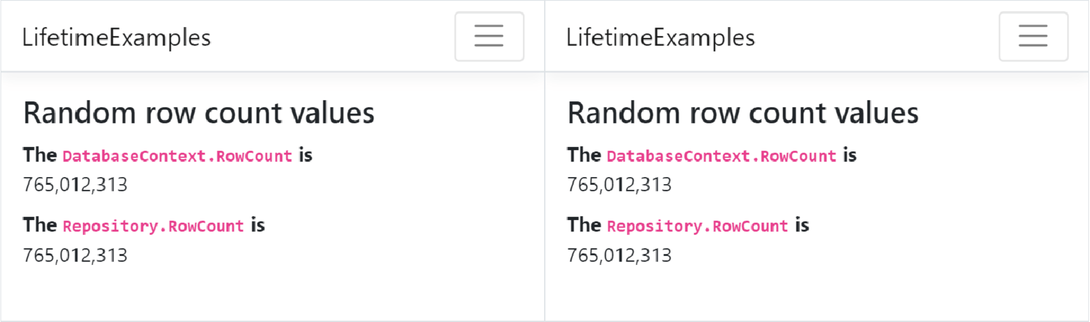
The singleton pattern is particularly useful for objects that are expensive to create, that contain data that must be shared across requests, or that don't hold state. The latter two points are important—any service registered as a singleton should be thread-safe.

> **WARNING** Singleton services must be thread-safe in a web application, as they'll typically be used by multiple threads during concurrent requests.

Let's consider what using singletons means for the row count example. I can update the registration of `DataContext` to be a singleton in `ConfigureServices`:

```
services.AddSingleton<DataContext>();
```

We can then call the RowCountModel Razor Page twice and observe the results in figure 10.15. You can see that every instance has returned the same value, indicating that all four instances of DataContext are the same single instance.



LifetimeExamples

Random row count values

The DatabaseContext.RowCount is
765,012,313

The Repository.RowCount is
765,012,313

LifetimeExamples

Random row count values

The DatabaseContext.RowCount is
765,012,313

The Repository.RowCount is
765,012,313

Figure 10.15 Any service registered as a singleton will always return the same instance. Consequently, all the calls to RowCount return the same value, both within a request and between requests.

Singletons are convenient for objects that need to be shared or that are immutable and expensive to create. A caching service should be a singleton, as all requests need to share it. It must be thread-safe though. Similarly, you might register a settings object loaded from a remote server as a singleton if you load the settings once at startup and reuse them through the lifetime of your app.

On the face of it, choosing a lifetime for a service might not seem too tricky, but there's an important "gotcha" that can come back to bite you in subtle ways, as you'll see shortly.

10.3.4 Keeping an eye out for captured dependencies

Imagine you're configuring the lifetime for the DataContext and Repository examples. You think about the suggestions I've provided and decide on the following lifetimes:

- DataContext—*Scoped*, as it should be shared for a single request
- Repository—*Singleton*, as it has no state of its own and is thread-safe, so why not?

WARNING This lifetime configuration is to explore a bug—don't use it in your code or you'll experience a similar problem!

Unfortunately, you've created a *captured dependency* because you're injecting a *scoped* object, DataContext, into a *singleton*, Repository. As it's a singleton, the same Repository instance is used throughout the lifetime of the app, so the DataContext that was injected into it will *also* hang around, *even though a new one should be used with every request*. Figure 10.16 shows this scenario, where a new instance of DataContext is

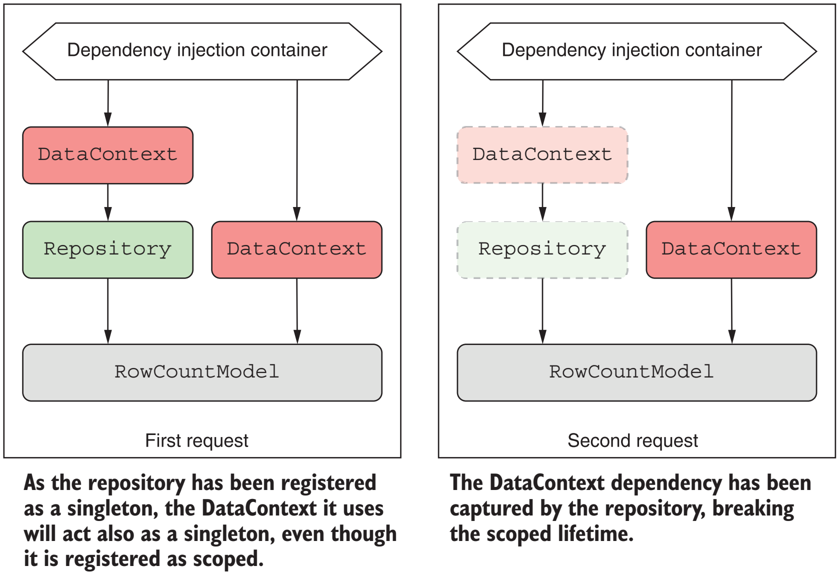

Dependency injection container
DataContext
Repository
DataContext
RowCountModel
First request
As the repository has been registered as a singleton, the DataContext it uses will act also as a singleton, even though it is registered as scoped.
Dependency injection container
DataContext
Repository
DataContext
RowCountModel
Second request
The DataContext dependency has been captured by the repository, breaking the scoped lifetime.

Figure 10.16 `DataContext` **is registered as a scoped dependency, but** `Repository` **is a singleton. Even though you expect a new** `DataContext` **for every request,** `Repository` ***captures*** **the injected** `DataContext` **and causes it to be reused for the lifetime of the app.**

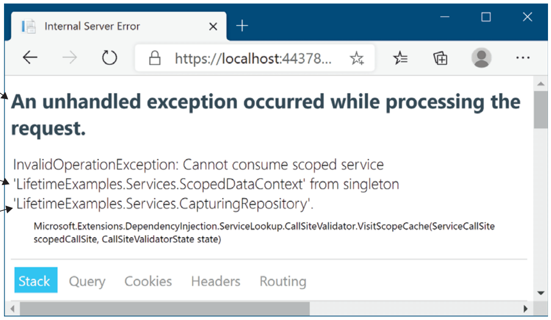
created for each scope, but the instance inside `Repository` hangs around for the lifetime of the app.

Captured dependencies can cause subtle bugs that are hard to root out, so you should always keep an eye out for them. These captured dependencies are relatively easy to introduce, so always think carefully when registering a singleton service.

> **WARNING** A service should only use dependencies with a lifetime longer than or equal to the lifetime of the service. A service registered as a singleton can only safely use singleton dependencies. A service registered as scoped can safely use scoped or singleton dependencies. A transient service can use dependencies with any lifetime.

At this point, I should mention that there's one glimmer of hope in this cautionary tale. ASP.NET Core automatically checks for these kinds of captured dependencies and will throw an exception on application startup if it detects them, as shown in figure 10.17.

This scope validation check has a performance impact, so by default it's only enabled when your app is running in a development environment, but it should help you catch most issues of this kind. You can enable or disable this check regardless of environment by setting the `ValidateScopes` option when creating your `HostBuilder` in Program.cs, as shown in listing 10.20.



An unhandled exception occurred while processing the request.

InvalidOperationException: Cannot consume scoped service 'LifetimeExamples.Services.ScopedDataContext' from singleton 'LifetimeExamples.Services.CapturingRepository'.

Microsoft.Extensions.DependencyInjection.ServiceLookup.CallSiteValidator.VisitScopeCache(ServiceCallSite scopedCallSite, CallSiteValidatorState state)

In a development environment, you will get an exception when the DI container detects a captured dependency.

The exception message describes which service was captured...

...and which service captured the dependency.

Figure 10.17 When `ValidateScopes` is enabled, the DI container will throw an exception when it creates a service with a captured dependency. By default, this check is only enabled for development environments.

Listing 10.20 Setting the `ValidateScopes` property to always validate scopes

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            })
            .UseDefaultServiceProvider(options =>
            {
                options.ValidateScopes = true;
                options.ValidateOnBuild = true;
            });
}
```

The default builder sets ValidateScopes to validate only in development environments.

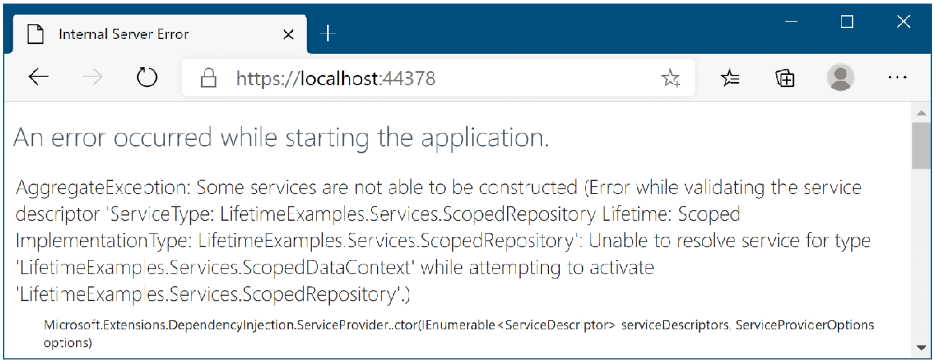
You can override the validation check with the UseDefaultServiceProvider extension.

Setting this to true will validate scopes in all environments. This has performance implications.

ValidateOnBuild checks that every registered service has all its dependencies registered.

Listing 10.20 shows another setting you can enable, `ValidateOnBuild`, that goes one step further. When it's enabled, the DI container checks on application startup that it has dependencies registered for every service it needs to build. If it doesn't, it throws an exception, as shown in figure 10.18, letting you know about the misconfiguration. This also has a performance impact, so it's only enabled in development environments by default, but it's very useful for pointing out any missed service registrations[6].

[6] Unfortunately, the container can't catch everything. For a list of caveats and exceptions, see this post from my blog: http://mng.bz/QmwG.



An error occurred while starting the application.

AggregateException: Some services are not able to be constructed (Error while validating the service descriptor 'ServiceType: LifetimeExamples.Services.ScopedRepository Lifetime: Scoped ImplementationType: LifetimeExamples.Services.ScopedRepository': Unable to resolve service for type 'LifetimeExamples.Services.ScopedDataContext' while attempting to activate 'LifetimeExamples.Services.ScopedRepository'.)

```
    Microsoft.Extensions.DependencyInjection.ServiceProvider..ctor(IEnumerable<ServiceDescriptor> serviceDescriptors, ServiceProviderOptions
    options)
```

Figure 10.18 When `ValidateOnBuild` is enabled, the DI container will check on app startup that it can create all of the registered services. If it finds a service it can't create, it throws an exception. By default, this check is only enabled for development environments.

With that, you've reached the end of this introduction to DI in ASP.NET Core. You now know how to add framework services to your app using `Add*` extension methods like `AddRazorPages()`, as well as how to register your own services with the DI container. Hopefully that will help you keep your code loosely coupled and easy to manage.

In the next chapter we'll look at the ASP.NET Core configuration model. You'll see how to load settings from a file at runtime, how to store sensitive settings safely, and how to make your application behave differently depending on which machine it's running on. We'll even use a bit of DI; it gets everywhere in ASP.NET Core!

Summary

- Dependency injection is baked into the ASP.NET Core framework. You need to ensure your application adds all the framework's dependencies in `Startup` or you will get exceptions at runtime when the DI container can't find the required services.
- The dependency graph is the set of objects that must be created in order to create a specific requested "root" object. The DI container handles creating all these dependencies for you.
- You should aim to use explicit dependencies over implicit dependencies in most cases. ASP.NET Core uses constructor arguments to declare explicit dependencies.
- When discussing DI, the term *service* is used to describe any class or interface registered with the container.
- You register services with a DI container so it knows which implementation to use for each requested service. This typically takes the form of "for interface X, use implementation Y."

- The DI or IoC container is responsible for creating instances of services. It knows how to construct an instance of a service by creating all the service's dependencies and passing these in the service constructor.
- The default built-in container only supports constructor injection. If you require other forms of DI, such as property injection, you can use a third-party container.
- You must register services with the container by calling `Add*` extension methods on `IServiceCollection` in `ConfigureServices` in `Startup`. If you forget to register a service that's used by the framework or in your own code, you'll get an `InvalidOperationException` at runtime.
- When registering your services, you describe three things: the service type, the implementation type, and the lifetime. The service type defines which class or interface will be requested as a dependency. The implementation type is the class the container should create to fulfill the dependency. The lifetime is how long an instance of the service should be used for.
- You can register a service using generic methods if the class is concrete and all its constructor arguments are registered with the container or have default values.
- You can provide an instance of a service during registration, which will register that instance as a singleton. This can be useful when you already have an instance of the service available.
- You can provide a lambda factory function that describes how to create an instance of a service with any lifetime you choose. You can use this approach when your services depend on other services that are only accessible once your application is running.
- Avoid calling `GetService()` in your factory functions if possible. Instead, favor constructor injection—it's more performant, as well as being simpler to reason about.
- You can register multiple implementations for a service. You can then inject `IEnumerable<T>` to get access to all the implementations at runtime.
- If you inject a single instance of a multiple-registered service, the container injects the last implementation registered.
- You can use the `TryAdd*` extension methods to ensure that an implementation is only registered if no other implementation of the service has been registered. This can be useful for library authors to add default services while still allowing consumers to override the registered services.
- You define the lifetime of a service during DI service registration. This dictates when a DI container will reuse an existing instance of the service to fulfill service dependencies and when it will create a new one.
- The transient lifetime means that every single time a service is requested, a new instance is created.

- The scoped lifetime means that within a scope all requests for a service will give you the same object. For different scopes, you'll get different objects. In ASP.NET Core, each web request gets its own scope.
- You'll always get the same instance of a singleton service, no matter which scope.
- A service should only use dependencies with a lifetime longer than or equal to the lifetime of the service.